Name

intro – introduction to the Network Computing System's (NCS) library routines

Description

This section describes the NCS library routines.

NOTE

The Title, Name, and See Also sections of the NCS reference pages do not contain the dollar ($) sign in the command names and library routines. The actual NCS commands and library routines do contain the dollar ($) sign.

The NCS commands and library routines are as follows:

- Error Text Database Operations (`error_$`)
- Interface to the Location Broker (`lb_$`)
- Fault Management (`pfm_$`)
- Program Management (`pgm_$`)
- Interface to the Remote Procedure Call Runtime Library (`rpc_$`)
- Remote Remote Procedure Call Interface (`rrpc_$`)
- Operations on Socket Addresses (`socket_$`)
- Operations on Universal Unique Identifiers (`uuid_$`)

Error Text Database Operations

The error text database operations use the `error_$c_get_text` and `error_$c_text` library routines to convert status codes into textual error messages. The runtime library reports operational problems back to the application following a call by setting the 'all' field of the **status_$t** structure. A value of **status_$ok** indicates that no errors were detected. Any other value implies that a problem occurred. The **status_$t** structure and the `error_$` routines can be used to display a textual representation of the error condition.

Data Types
This section describes the data types used in `error_$` routines.

The `error_$` routines take as input a status code in **status_$t** format.

status_$t A status code. Most of the NCS routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
   long all;
}
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
 struct {
    unsigned fail : 1,
             subsys : 7,
             modc : 8;
     short    code;
 } s;
 long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys
This indicates the subsystem that encountered the error.

modc
This indicates the module that encountered the error.

code
This is a signed number that identifies the type of error that occurred.

Interface To The Location Broker

The `lb_$` library routines implement the programmatic interface to the Location Broker Client Agent. The file `/usr/include/idl/c/glb.h` defines this interface.

External Variables
This section describes the external variable used in `lb_$` routines.

uuid_$nil An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Constants
This section describes constants used in `lb_$` routines.

lb_$default_lookup_handle
Used as an input in Location Broker lookup routines. Specifies that a lookup is to start searching at the beginning of the database.

lb_$server_flag_local Used in the **flags** field of an **lb_$entry_t** variable. Specifies that an entry is to be registered only in the Local Location Broker (LLB) database. See the description of **lb_$server_flag_t** in the Data Types section.

status_$ok A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful.

Data Types
This section describes data types used in `lb_$` routines.

lb_$entry_t An identifier for an object, a type, an interface, and the socket address used to access a server exporting the interface to the object. The **lb_$entry_t** type is defined as follows:

```
typedef struct lb_$entry_t lb_$entry_t;
struct lb_$entry_t {
    uuid_$t object;
    uuid_$t obj_type;
    uuid_$t obj_interface;
    lb_$server_flag_t flags;
    ndr_$char annotation[64];
    ndr_$ulong_int saddr_len;
    socket_$addr_t saddr;
};
```

object A **uuid_$t**. The UUID for the object. Can be **uuid_$nil** if no object is associated.

obj_type A **uuid_$t**. The UUID for the type of the object. Can be **uuid_$nil** if no type is associated.

obj_interface A **uuid_$t**. The UUID for the interface. Can be **uuid_$nil** if no interface is associated.

flags An **lb_$server_flag_t**. Must be 0 or **lb_$server_flag_local**. A value of 0 specifies that the entry is to be registered in both the Local Location Broker (LLB) and global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

annotation A 64-character array. User-defined textual annotation.

saddr_len A 32-bit integer. The length of the **saddr** field.

saddr A **socket_$addr_t**. The socket address of the server.

lb_$lookup_handle_t A 32-bit integer used to specify the location in the database at which a Location Broker lookup operation will start.

lb_$server_flag_t A 32-bit integer used to specify the Location Broker databases in which an entry is to be registered. A value of 0 specifies registration in both the Local Location Broker (LLB) and Global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

socket_$addr_t A socket address record that uniquely identifies a socket.

status_$t A status code. Most of the NCS routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all	All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.
fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.
subsys	This indicates the subsystem that encountered the error.
modc	This indicates the module that encountered the error.
code	This is a signed number that identifies the type of error that occurred.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time.

Example
The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
    &num_results, &matrix_results, &st);
```

Fault Management

The `pfm_$` routines allow programs to manage signals, faults, and exceptions by establishing clean-up handlers.

A clean-up handler is a piece of code that ensures a program terminates gracefully when it receives a fatal error. A clean-up handler begins with a `pfm_$cleanup` call, and usually ends with a call to `pfm_$signal` or `pgm_$exit`, though it can also simply continue back into the program after the clean-up code.

A clean-up handler is not entered until all fault handlers established for a fault have returned. If there is more than one established clean-up handler for a program, the most recently established clean-up handler is entered first, followed by the next most recently established clean-up handler, and so on to the first established clean-up handler if necessary.

There is a default clean-up handler invoked after all user-defined handlers have completed. It releases any resources still held by the program, before returning control to the process that invoked it.

Constants

pfm_$init_signal_handlers
: A constant used as the *flags* parameter to `pfm_$init`, causing C signals to be intercepted and converted to PFM signals.

Data Types

This section describes the data typed used in pfm_$ routines.

pfm_$cleanup_rec
: A record type for passing process context among clean-up handler routines. It is an opaque data type.

status_$t
: A status code. Most of the NCS routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all
: All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail
: If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys
: This indicates the subsystem that encountered the error.

modc	This indicates the module that encountered the error.
code	This is a signed number that identifies the type of error that occurred.

Program Management

The NCS software products contain a portable version of the `pgm_$exit` routine. The include file for the PFM interface (see the Syntax section of the `pfm`(3ncs) reference pages) contains a declaration for this routine.

Interface To The Remote Procedure Call

The `rpc_$` library routines implement the NCS Remote Procedure Call (RPC) mechanism.

The `rpc_` interface is defined by the file `/usr/include/idl/rpc.idl`.

Most of the `rpc_$` routines can be used only by clients or only by servers. This aspect of their usage is specified at the beginning of each routine description, in the Name section.

External Variables

This section describes the external variable used in **rpc_$** routines.

uuid_$nil	An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Constants

This section describes constants used in **rpc_$** routines.

rpc_$mod	A module code indicating the RPC module.
status_$ok	A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful. See the description of the **status_$t** type.
rpc_$unbound_port	A port number indicating to the RPC runtime library that no port is specified. Identical to **socket_$unspec_port**.

The following 16-bit-integer constants are used to specify the communications protocol address families in **socket_$addr_t** structures. Note that several of the **rpc_$** and **socket_$** calls use the 32-bit-integer equivalents of these values.

socket_$unspec	Address family is unspecified.
socket_$internet	Internet Protocols (IP).

Data Types

This section describes data types used in **rpc_$** routines.

handle_t	An RPC handle.
rpc_$epv_t	An entry point vector (EPV). An array of **rpc_$server_stub_t**, pointers to server stub procedures.
rpc_$generic_epv_t	An entry point vector (EPV). An array of **rpc_$generic_server_stub_t**, pointers to generic server stub procedures.

rpc_$if_spec_t An RPC interface specifier. This opaque data type contains information about an interface, including its UUID, the current version number, any well-known ports used by servers that export the interface, and the number of operations in the interface.

rpc_$mgr_epv_t An entry point vector (EPV). An array of pointers to manager procedures.

rpc_$shut_check_fn_t A pointer to a function. If a server supplies this function pointer to **rpc_$allow_remote_shutdown**, the function will be called when a remote shutdown request arrives, and if the function returns true, the shutdown is allowed. The following C definition for **rpc_$shut_check_fn_t** illustrates the prototype for this function:

```
typedef boolean (*rpc_$shut_check_fn_t) (
    handle_t h,
    status_$t *st)
```

The handle argument can be used to determine information about the remote caller.

socket_$addr_t A socket address record that uniquely identifies a socket.

status_$t A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

	modc	This indicates the module that encountered the error.
	code	This is a signed number that identifies the type of error that occurred.
uuid_$t		A 128-bit value that uniquely identifies an object, type, or interface for all time.

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
h = rpc_$alloc_handle (&acme_pay_id, socket_$internet, &st);
```

Remote Remote Procedure Call Interface

The `rrpc_$` library routines enable a client to request information about a server or to shut down a server.

The `rrpc_` interface is defined by the file `/usr/include/idl/rrpc.idl`.

Constants

This section describes constants used in **rrpc_$** calls.

The **rrpc_$sv** constants are indices for elements in an **rrpc_$stat_vec_t** array. Each element is a 32-bit integer representing a statistic about a server. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in	The number of calls processed by the server.
rrpc_$sv_rcvd	The number of packets received by the server.
rrpc_$sv_sent	The number of packets sent by the server.
rrpc_$sv_calls_out	The number of calls made by the server.
rrpc_$sv_frag_resends	The number of fragments sent by the server that duplicated previous sends.
rrpc_$sv_dup_frags_rcvd	The number of duplicate fragments received by the server.

status_$ok	A constant used to check status. If a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

Data Types

This section describes data types used in **rpc_$** routines.

handle_t	An RPC handle.
rrpc_$interface_vec_t	An array of **rpc_$if_spec_t**, RPC interface specifiers.
rrpc_$stat_vec_t	An array of 32-bit integers, indexed by **rrpc_$sv** constants, representing statistics about a server.
rpc_$if_spec_t	An RPC interface specifier. An opaque data type containing information about an interface, including the UUID, the

version number, the number of operations in the interface, and any well-known ports used by servers that export the interface, and any well-known ports used by servers that export the interface. Applications may need to access two members of **rpc_$if_spec_t**:

id A **uuid_$t** indicating the interface UUID.

vers An unsigned 32-bit integer indicating the interface version.

Operations on Socket Addresses

The `socket_$` library routines manipulate socket addresses. Unlike the routines that operating systems such as BSD UNIX provide, the `socket_$` routines operate on addresses of any protocol family.

The file `/usr/include/idl/socket.idl` defines the `socket_` interface.

Constants

This section describes constants used in **socket_$** routines.

The **socket_$eq** constants are flags indicating the fields to be compared in a **socket_$equal** call.

socket_$eq_hostid Indicates that the host IDs are to be compared.

socket_$eq_netaddr Indicates that the network addresses are to be compared.

socket_$eq_port Indicates that the port numbers are to be compared.

socket_$eq_network Indicates that the network IDs are to be compared.

socket_$unspec_port A port number indicating to the RPC runtime library that no port is specified.

The following 16-bit-integer constants are values for the **socket_$addr_family_t** type, used to specify the address family in a **socket_$addr_t** structure. Note that several of the **rpc_$** and **socket_$** routines use the 32-bit-integer equivalents of these values.

socket_$unspec Address family is unspecified.

socket_$internet Internet Protocols (IP).

status_$ok A constant used to check status. If a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

Data Types

This section describes data types used in **socket_$** routines.

socket_$addr_family_t
An enumerated type for specifying an address family. The Constants section lists values for this type.

socket_$addr_list_t An array of socket addresses in **socket_$addr_t** format.

socket_$addr_t A structure that uniquely identifies a socket address. This structure consists of a **socket_$addr_family_t** specifying an address family and 14 bytes specifying a socket address.

socket_$host_id_t A structure that uniquely identifies a host. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a host.

socket_$len_list_t An array of unsigned 32-bit integers, the lengths of socket addresses in a **socket_$addr_list_t**.

socket_$local_sockaddr_t
An array of 50 characters, used to store a socket address in a format native to the local host.

socket_$net_addr_t A structure that uniquely identifies a network address. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a network address. It contains both a host ID and a network ID.

socket_$string_t An array of 100 characters, used to store the string representation of an address family or a socket address.

The string representation of an address family is a textual name such as **dds**, **ip**, or **unspec**.

The string representation of a socket address has the format *family*:*host*[*port*], where *family* is the textual name of an address family, *host* is either a textual host name or a numeric host ID preceded by a #, and *port* is a port number.

status_$t A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.
subsys	This indicates the subsystem that encountered the error.
modc	This indicates the module that encountered the error.
code	This is a signed number that identifies the type of error that occurred.

Operations On Universal Unique Identifiers

The `uuid_$` library routines operate on UUIDs (Universal Unique Identifiers).

The `uuid_` interface is defined by the file `/usr/include/idl/uuid.idl`.

The completion status. `/usr/include/idl/uuid.idl`

External Variables
This section describes external variables used in **uuid_$** routines.

uuid_$nil
An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Data Types
This section describes data types used in **uuid_$** routines.

status_$t A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all	All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.
fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$string_t
A string of 37 characters (including a null terminator) that is an ASCII representation of a UUID. The format is *ccccccccccc.ff.h1.h2.h3.h4.h5.h6.h7*, where *ccccccccccc* is the timestamp, *ff* is the address family, and *h1* ... *h7* are the 7 bytes of host identifier. Each character in these fields is a hexadecimal digit.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. The **uuid_$t** type is defined as follows:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

time_high
The high 32 bits of a 48-bit unsigned time value which is the number of 4-microsecond intervals that have passed between 1 January 1980 00:00 GMT and the time of UUID creation.

time_low
The low 16 bits of the 48-bit time value.

reserved
16 bits of reserved space.

family
8 bits identifying an address family.

host 7 bytes identifying the host on which the UUID was created. The format of this field depends on the address family.

Example
The following routine returns as `foo_uuid` the UUID corresponding to the character-string representation in `foo_uuid_rep`:

```
uuid_$decode (foo_uuid_rep, &foo_uuid, &status);
```

Name

error_c_get_text – return subsystem, module, and error texts for a status code

Syntax

void error_$c_get_text(*status, subsys, subsysmax, module, modulemax, error, errormax*)

status_$t *status*;
char **subsys*;
long *subsysmax*;
char **module*;
long *modulemax*;
char **error*;
long *errormax*;

Arguments

status	A status code in **status_$t** format.
subsys	A character string. The subsystem represented by the status code.
subsysmax	The maximum number of bytes to be returned in *subsys*.
module	A character string. The module represented by the status code.
modulemax	The maximum number of bytes to be returned in *module*.
error	A character string. The error represented by the status code.
errormax	The maximum number of bytes to be returned in *error*.

Description

The `error_$c_get_text` routine returns predefined text strings that describe the subsystem, the module, and the error represented by a status code. The strings are null terminated. See the `intro`(3ncs) reference page which lists all of the possible diagnostics that could be returned in `status.all`.

Files

`/usr/lib/stcode.db`

See Also

intro(3ncs)

error_c_text (3ncs)

Name

error_c_text – return an error message for a status code

Syntax

void error_$c_text(*status, message, messagemax*)
status_$t *status*;
char **message*;
int *messagemax*;

Arguments

status	A status code in **status_$t** format.
message	A character string. The error message represented by the status code.
messagemax	The maximum number of bytes to be returned in *message*.

Description

The `error_$c_text` routine returns a null terminated error message for reporting the completion status of a routine. The error message is composed from predefined text strings that describe the subsystem, the module, and the error represented by the status code. See the `intro`(3ncs) reference page which lists all of the possible diagnostics that could be returned in `status.all`.

Files

`/usr/lib/stcode.db`

See Also

intro(3ncs)

Name

lb_lookup_interface – look up information about an interface in the Global Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$lookup_interface(obj_interface, lookup_handle, max_num_results,
                          num_results, results, status)
uuid_$t *obj_interface;
lb_$lookup_handle_t *lookup_handle;
unsigned long max_num_results;
unsigned long * num_results;
lb_$entry_t results[ ];
status_$t *status;
```

Arguments

obj_interface	The UUID of the interface being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status.

Description

The `lb_$lookup_interface` routine returns GLB database entries whose *obj_interface* fields match the specified interface. It returns information about objects that can be accessed through that interface.

The `lb_$lookup_interface` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
                      &num_results, &matrix_results, &st);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

/usr/include/idl/c/glb.h

See Also

intro(3ncs), lb_lookup_object(3ncs), lb_lookup_range(3ncs), lb_lookup_type(3ncs)

lb_lookup_object(3ncs)

Name

lb_lookup_object – look up information about an object in the Global Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$lookup_object(object, lookup_handle, max_num_results,
                       num_results, results, status)
uuid_$t *object;
lb_$lookup_handle_t *lookup_handle;
unsigned long max_num_results;
unsigned long * num_results;
lb_$entry_t results[ ];
status_$t *status;
```

Arguments

object	The UUID of the object being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_object` routine returns GLB database entries whose *object* field matches the specified object UUID.

The `lb_$lookup_object` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement, looks up GLB database entries for the object identified by `bank_id`:

```
lb_$lookup_object(&bank_id, &lookup_handle,
                        MAX_LOCS, &n_locs, bank_loc, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object_local(3ncs), lb_lookup_range(3ncs), lb_lookup_type(3ncs)

Name

lb_lookup_object_local – look up information about an object in a Local Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$lookup_object_local(object, location, location_length, lookup_handle
                             max_num_results, num_results, results, status)
uuid_$t *object;
socket_$addr_t *location;
unsigned long location_length;
lb_$lookup_handle_t *lookup_handle;
unsigned long max_num_results;
unsigned long *num_results;
lb_$entry_t results[ ];
status_$t *status;
```

Arguments

object	The UUID of the object being looked up.
location	The location of the LLB database to be searched. The socket address must specify the network address of a host. However, the port number in the socket address is ignored, and the lookup request is always sent to the LLB port.
location_length	The length, in bytes, of the socket address specified by the location field.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.

results — An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_object_local` routine searches the specified LLB database and returns all entries whose *object* field matches the specified object.

The `lb_$lookup_object_local` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information about the object **locobj**. Since there is only one entry on any host, the routine will return at most one result:

```
lb_$lookup_object_local (&locobj_id, &location, location_length,
                         &lookup_handle, 1, &num_results,
                         &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid — The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy — The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered — The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range`

routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_range(3ncs)

lb_lookup_range (3ncs)

Name

lb_lookup_range – look up information in a Global Location Broker or Local Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$lookup_range(object, obj_type, obj_interface, location,
                      location_length, lookup_handle, max_num_results,
                      num_results, results, status)
uuid_$t *object;
uuid_$t *obj_type;
uuid_$t *obj_interface;
socket_$addr_t *location;
unsigned long location_length;
lb_$lookup_handle_t *lookup_handle;
unsigned long max_num_results;
unsigned long *num_results;
lb_$entry_t results[ ];
status_$t *status);
```

Arguments

object	The UUID of the object being looked up.
obj_type	The UUID of the type being looked up.
obj_interface	The UUID of the interface being looked up.
location	The location of the database to be searched. If the value of *location_length* is 0, the GLB database is searched. Otherwise, the LLB database at the host specified by *location* is searched; in this case, the port number in the socket address is ignored, and the lookup request is sent to the LLB port.
location_length	The length, in bytes, of the socket address specified by the *location* field. A value of 0 indicates that the GLB database is to be searched.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.

max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_range` routine returns database entries whose *object, obj_type,* and *obj_interface* fields match the specified values. A value of **uuid_$nil** in any of these input parameters acts as a wildcard and will match any value in the corresponding entry field. You can specify wildcards in any combination of these parameters.

The `lb_$lookup_range` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about servers that export the **matrix** interface for any objects of type **array.** The variable **glb** is defined elsewhere as a null pointer.

```
lb_$lookup_range(&uuid_$nil, &array_id, &matrix_id, glb, 0,
                 &lookup_handle, max_results,
                 &num_results, results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid	The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object(3ncs), lb_lookup_object_local(3ncs), lb_lookup_type(3ncs)

Name

lb_lookup_type – look up information about a type in the Global Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$lookup_type(obj_type, lookup_handle, max_num_results,
                                    num_results, results, status)
uuid_$t *obj_type;
lb_$lookup_handle_t *lookup_handle;
unsigned long max_num_results;
unsigned long *num_results;
lb_$entry_t results[ ];
status_$t *status;
```

Arguments

obj_type	The UUID of the type being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

lb_lookup_type(3ncs)

Description

The `lb_$lookup_type` routine returns GLB database entries whose *obj_type* fields match the specified type. It returns information about all objects of that type and about all interfaces to each of these objects.

The `lb_$lookup_type` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about the type **array** :

```
lb_$lookup_type (&array_id, &lookup_handle, max_results,
                    &num_results, &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist, and the Location Broker cannot create it.

2. The database exists, but the Location Broker cannot access it.

3. The GLB entry table is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object(3ncs), lb_lookup_range(3ncs)

lb_register (3ncs)

Name

lb_register – register an object and an interface with the Location Broker

Syntax

```
#include <idl/c/lb.h>

void lb_$register(object, obj_type, obj_interface, flags, annotation,
                  location, location_length, entry, status)
uuid_$t *object;
uuid_$t *obj_type;
uuid_$t *obj_interface;
lb_$server_flag_t flags;
unsigned char annotation[64];
socket_$addr_t *location;
unsigned long location_length;
lb_$entry_t *entry;
status_$t *status;
```

Arguments

object	The UUID of the object being registered.
obj_type	The UUID of the type of the object being registered.
obj_interface	The UUID of the interface being registered.
flags	Must be either **lb_$server_flag_local** (specifying registration with only the LLB at the local host) or 0 (specifying registration with both the LLB and the GLB).
annotation	A character array used only for informational purposes. This field can contain a textual description of the object and the interface. For proper display by the `lb_admin` tool, the *annotation* should be terminated by a null character.
location	The socket address of the server that exports the interface to the object.
location_length	The length, in bytes, of the socket address specified by the *location* field.
entry	A copy of the entry that was entered in the Location Broker database.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$register` routine registers with the Location Broker an interface to an object and the location of a server that exports that interface. This routine replaces any existing entry in the Location Broker database that matches *object, obj_type, obj_interface,* and both the address family and host in *location;* if no such entry exists, the routine adds a new entry to the database.

If the *flags* parameter is `lb_$server_flag_local`, the entry is registered only in the LLB database at the host where the call is issued. Otherwise, the flag should be 0 to register with both the LLB and the GLB databases.

Examples

The following statement registers the `bank` interface to the object identified by `bank_id`:

```
lb_$register (&bank_id, &bank_$uuid, &bank_$if_spec.id, 0,
                  BankName, &saddr, slen, &entry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$update_failed The Location Broker was unable to register the entry.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist, and the Location Broker cannot create it.

2. The database exists, but the Location Broker cannot access it.

3. The GLB entry table is full.

lb_$server_unavailable The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_unregister(3ncs)

lb_unregister (3ncs)

Name

lb_unregister – remove an entry from the Location Broker database

Syntax

```
#include <idl/c/lb.h>

void lb_$unregister(entry, status)
lb_$entry_t *entry;
status_$t *status;
```

Arguments

entry	The entry being removed from the Location Broker database.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `lb_$unregister` routine removes from the Location Broker database the entry that matches *entry*. The value of *entry* should be identical to that returned by the `lb_$register` routine when the database entry was created. However, `lb_$unregister` does not compare all of the fields in *entry*, the **annotation** field, and the port number in the **saddr** field.

This routine removes the entry from the LLB database on the local host (the host that issues the routine). If the **flags** field of *entry* is equal to 0, it removes the entry from the GLB database. If the **flags** field is equal to **lb_$server_flag_local,** it deletes only the LLB entry.

Examples

The following statement unregisters the entry specified by `BankEntry`, which was obtained from a previous `lb_$register` routine:

```
lb_$unregister (&BankEntry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid	The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.
lb_$database_busy	The Location Broker database is currently in use in an incompatible manner.
lb_$not_registered	The Location Broker does not have any entries that match

the criteria specified in the unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database.

lb_$update_failed The Location Broker was unable to register or unregister the entry.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_register(3ncs)

pfm_cleanup (3ncs)

Name

pfm_cleanup – establish a clean-up handler

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

status_$t pfm_$cleanup(cleanup_record)
pfm_$cleanup_rec *cleanup_record;
```

Arguments

cleanup_record	A record of the context when `pfm_$cleanup` is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by `pfm_$rls_cleanup` and `pfm_$reset_cleanup` to restore the context of the calling process at the clean-up handler entry point.

Description

The `pfm_$cleanup` routine establishes a clean-up handler that is executed when a fault occurs. A clean-up handler is a piece of code executed before a program exits when a signal is received by the process. The clean-up handler begins where `pfm_$cleanup` is called; the `pfm_$cleanup` routine registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to `pfm_$cleanup`.

There can be more than one clean-up handler in a program. Multiple clean-up handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first clean-up handler. The system provides a default clean-up handler established at program invocation. The default clean-up handler is always called last, just before a program exits, and releases any system resources still held, before returning control to the process that invoked the program.

Diagnostics

When called to establish a clean-up handler, `pfm_$cleanup` returns the status **pfm_$cleanup_set** to indicate the clean-up handler was successfully established. When the clean-up handler is entered in response to a fault signal, `pfm_$cleanup` effectively returns the value of the fault that triggered the handler.

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order	Attempted to release a clean-up handler out of order.
pfm_$cleanup_not_found	There is no pending clean-up handler.
pfm_$cleanup_set	A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

NOTE

Clean-up handler code runs with asynchronous faults inhibited. When `pfm_$cleanup` returns something other than **pfm_$cleanup_set** indicating that a fault has occurred, there are four possible ways to leave the clean-up code:

- The program can call `pfm_$signal` to start the next clean-up handler with a different fault signal.
- The program can call `pgm_$exit` to start the next clean-up handler with the same fault signal.
- The program can continue with the code following the clean-up handler. It should generally call `pfm_$enable` to reenable asynchronous faults. Execution continues from the end of the clean-up handler code; it does not resume where the fault signal was received.
- The program can reestablish the handler by calling `pfm_$reset_cleanup` before proceeding.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_signal(3ncs)

pfm_enable(3ncs)

Name

pfm_enable – enable asynchronous faults

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$enable()
```

Description

The `pfm_$enable` routine enables asynchronous faults after they have been inhibited by a routine to `pfm_$inhibit`; `pfm_$enable` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable` returns, there can be at most one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit` and `pfm_$enable`, the process receives the first asynchronous fault received while faults were inhibited.

See Also

intro(3ncs), pfm_enable_faults(3ncs), pfm_inhibit(3ncs)

Name

pfm_enable_faults – enable asynchronous faults

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$enable_faults()
```

Description

The `pfm_$enable_faults` routine enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit_faults`; `pfm_$enable_faults` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable_faults` returns, there can be at most one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit_faults` and `pfm_$enable_faults`, the process receives the first asynchronous fault received while faults were inhibited.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable(3ncs), pfm_inhibit_faults(3ncs)

pfm_inhibit(3ncs)

Name

pfm_inhibit – inhibit asynchronous faults

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$inhibit()
```

Description

The `pfm_$inhibit` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This routine has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable(3ncs), pfm_inhibit_fault(3ncs)

Name

pfm_inhibit_faults – inhibit asynchronous faults

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$inhibit_faults()
```

Description

The `pfm_$inhibit_faults` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit_faults` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable_faults(3ncs), pfm_inhibit(3ncs)

pfm_init(3ncs)

Name

pfm_init – initialize the PFM package

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$init(flags)
unsigned long flags;
```

Arguments

flags

pfm_init_signal_handlers
Currently the only valid flag value. A flag's variable must be set to contain this value or the call will perform no initialization . A call to **pfm_init_signal_handlers** causes C signals to be intercepted and converted to PFM signals. On ULTRIX and VMS systems, the signals intercepted are SIGINIT, SIGILL, SIGFPE, SIGTERM, SIGHUP, SIGQUIT, SIGTRAP, SIGBUS, SIGSEGV, and SIGSYS.

Description

The call to `pfm_$init()` establishes a default set of signal handlers for the routine. The call to `pfm_$init()` should be made prior to the application's use of all other runtime RPC routines. This enables the RPC runtime system to catch and report all fault and/or interrupt signals that may occur during normal operation. Additionally, the user may provide a fault processing clean-up handler for application-specific exit handling.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_cleanup(3ncs)

Name

pfm_reset_cleanup – reset a clean-up handler

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$reset_cleanup(cleanup_record, status)
pfm_$cleanup_rec *cleanup_record;
status_$t *status;
```

Arguments

cleanup_record	A record of the context at the clean-up handler entry point. It is supplied by `pfm_$cleanup`, when the clean-up handler if first established.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$reset_cleanup` routine reestablishes the clean-up handler last entered so that any subsequent errors enter it first. This procedure should only be used within clean-up handler code.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/c/pfm.h
```

See Also

intro(3ncs)

pfm_rls_cleanup (3ncs)

Name

pfm_rls_cleanup – release clean-up handlers

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$rls_cleanup(cleanup_record, status)
pfm_$cleanup_rec *cleanup_record;
status_$t *status;
```

Arguments

cleanup_record	The clean-up record for the first clean-up handler to release.
status	The completion status. If *status* is **pfm_$bad_rls_order**, it means that the caller attempted to release a clean-up handler before releasing all handlers established after it. This status is only a warning; the intended clean-up handler is released, along with all clean-up handlers established after it. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$rls_cleanup` routine releases the clean-up handler associated with *cleanup_record* and all clean-up handlers established after it.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

pfm_signal(3ncs)

Name

pfm_signal – signal the calling process

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$signal(fault_signal)
status_$t *fault_signal;
```

Arguments

fault_signal	A fault code.

Description

The `pfm_$signal` routine signals the fault specified by *fault_signal* to the calling process. It is usually called to leave clean-up handlers.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

NOTE

This routine does not return when successful.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

Name

pgm_exit – exit a program

Syntax

```
#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pgm_$exit()
```

Description

The `pgm_$exit` routine exits from the calling program and returns control to the process that invoked it. When `pgm_$exit` is called any files left open by the program are closed, any storage acquired is released, and asynchronous faults are reenabled if they were inhibited by the calling program.

The `pgm_$exit` routine always calls `pfm_$signal()` with a status of **status_$ok.**

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

rpc_alloc_handle (3ncs)

Name

rpc_alloc_handle – create an RPC handle (client only)

Syntax

#include <idl/c/rpc.h>

handle_t rpc_$alloc_handle(*object, family, status)*
uuid_$t **object*;
unsigned long *family*;
status_$t **status*;

Arguments

object	The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.
family	The address family to use in communications to access the object. Currently, only **socket_$ internet** is supported.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$alloc_handle` routine creates an unbound RPC handle that identifies a particular object but not a particular server or host.

If a remote procedure call is made using the unbound handle, it will effect a broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first responding server.

Examples

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
h = rpc_$alloc_handle (&acme_pay_id, socket_$internet, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure	The client was unable to get a response from the server.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$wrong_boot_time
The server boot time value maintained by the client does not correspond to the current server boot time. The server was probably rebooted while the client program was running.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_free_handle(3ncs), rpc_set_binding(3ncs)

rpc_allow_remote_shutdown (3ncs)

Name

rpc_allow_remote_shutdown – allow or disallow remote shutdown of a server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$allow_remote_shutdown(*allow, checkproc, status)*
unsigned long allow;
rpc_$shut_check_fn_t checkproc;
status_$t **status*;

Arguments

allow	A value indicating 'false' if zero, 'true' otherwise.
checkproc	A pointer to a Boolean function.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$allow_remote_shutdown` routine allows or disallows remote callers to shut down a server using `rrpc_$shutdown.`

By default, servers do not allow remote shutdown via `rrpc_$shutdown.` If a server calls `rpc_$allow_remote_shutdown` with *allow* true (not zero) and *checkproc* nil, then remote shutdown will be allowed. If *allow* is true and *checkproc* is not nil, then when a remote shutdown request arrives, the function denoted by *checkproc* is called and the shutdown is allowed if the function returns true. If *allow* is false (zero), remote shutdown is disallowed.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all.`

rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_shutdown(3ncs), rrpc_shutdown(3ncs)

rpc_bind(3ncs)

Name

rpc_bind – allocate an RPC handle and set its binding to a server (client only)

Syntax

```
#include <idl/c/rpc.h>

handle_t rpc_$bind(object, sockaddr, slength, status)
uuid_$t *object;
socket_$addr_t *sockaddr;
unsigned long slength;
status_$t *status;
```

Arguments

object	The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.
sockaddr	The socket address of the server.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$bind` routine creates a fully bound RPC handle that identifies a particular object and server. This routine is equivalent to an `rpc_$alloc_handle` routine followed by an `rpc_$set_binding` routine.

Examples

The following statement binds the binop client to the specified object and socket address. The **loc** parameter is the result of a previous call to `rpc_$name_to_sockaddr` which converted the host name and port number to a socket address.

```
rh = rpc_$bind (&uuid_$nil, &loc, llen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock	The RPC runtime library created a socket but was unable to bind it to a socket address.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_clear_binding(3ncs), rpc_clear_server_binding(3ncs), rpc_set_binding(3ncs)

rpc_clear_binding (3ncs)

Name

rpc_clear_binding – unset the binding of an RPC handle to a host and server (client only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$clear_binding(handle, status)
handle_t handle;
status_$t *status;
```

Arguments

handle The RPC handle whose binding is being cleared.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$clear_binding` routine removes any association between an RPC handle and a particular server and host, but it does not remove the association between the handle and an object. This routine saves the RPC handle so that it can be reused to access the same object, either by broadcasting or after resetting the binding to another server.

A remote procedure call made using an unbound handle is broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first server that responded.

The `rpc_$clear_binding` routine is the inverse of the `rpc_$set_binding` routine.

Examples

Clear the binding represented in *handle*:

```
rpc_$clear_binding (handle, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_clear_server_binding(3ncs), rpc_set_binding(3ncs)

rpc_clear_server_binding (3ncs)

Name

rpc_clear_server_binding – unset the binding of an RPC handle to a server (client only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$clear_server_binding(handle, status)
handle_t handle;
status_$t *status;
```

Arguments

handle	The RPC handle whose binding is being cleared.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$clear_server_binding` routine removes the association between an RPC handle and a particular server (that is, a particular port number), but does not remove the associations with an object and with a host (that is, a network address). This call replaces a fully bound handle with a bound-to-host handle. A bound-to-host handle identifies an object located on a particular host but does not identify a server exporting an interface to the object.

If a client uses a bound-to-host handle to make a remote procedure call, the call is sent to the Local Location Broker (LLB) forwarding port at the host identified by the handle. If the call's interface and the object identified by the handle are both registered with the host's LLB, the LLB forwards the request to the registering server. When the client RPC runtime library receives a response, it binds the handle to the server. Subsequent remote procedure calls that use this handle are then sent directly to the bound server's port.

The `rpc_$clear_server_binding` routine is useful for client error recovery when a server dies. The port that a server uses when it restarts is not necessarily the same port that it used previously; therefore, the binding that the client was using may not be correct. This routine enables the client to unbind from the dead server while retaining the binding to the host. When the client sends a request, the binding is automatically set to the server's new port.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/rpc.idl
/usr/include/idl/c/rpc.h
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_clear_binding(3ncs), rpc_set_binding(3ncs)

rpc_dup_handle(3ncs)

Name

rpc_dup_handle – make a copy of an RPC handle (client only)

Syntax

#include <idl/c/rpc.h>

handle_t rpc_$dup_handle(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle The RPC handle to be copied.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$dup_handle` routine returns a copy of an existing RPC handle. Both handles can then be used in the client program for concurrent multiple accesses to a binding. Because all duplicates of a handle reference the same data, an `rpc_$set_binding`, `rpc_$clear_binding`, or `rpc_$clear_server_binding` routine made on any one duplicate affects all duplicates. However, an RPC handle is not freed until `rpc_$free_handle` is called on all copies of the handle.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_free_handle(3ncs)

Name

rpc_free_handle – free an RPC handle (client only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$free_handle(handle, status)
handle_t handle;
status_$t *status;
```

Arguments

handle The RPC handle to be freed.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$free_handle` routine frees an RPC handle. This routine clears any association between the handle and a server or an object and releases the resources identified by the RPC handle. The client program cannot use a handle after it is freed.

Examples

The following statement frees a handle:

```
rpc_$free_handle (handle, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_dup_handle(3ncs)

rpc_inq_binding(3ncs)

Name

rpc_inq_binding – return the socket address represented by an RPC handle (client or server)

Syntax

#include <idl/c/rpc.h>

void rpc_$inq_binding(*handle, sockaddr, slength, status*)
handle_t *handle*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

handle	An RPC handle.
sockaddr	The socket address represented by *handle*.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$inq_binding` routine enables a client to determine the socket address, and therefore the server, identified by an RPC handle. It is useful when a client uses an unbound handle in a remote procedure call and wishes to determine the particular server that responded to the call.

Examples

The Location Broker administrative tool, `lb_admin`, uses the following statement to determine the GLB that last responded to a lookup request:

```
rpc_$inq_binding(lb_$handle, &global_broker_addr,
                 &global_broker_addr_len, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.
rpc_$unbound_handle	The handle is not bound and does not represent a particular host address. Returned by `rpc_$inq_binding`.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_set_binding(3ncs)

rpc_inq_object(3ncs)

Name

rpc_inq_object – return the object UUID represented by an RPC handle (client or server)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$inq_object(handle, object, status)
handle_t handle;
uuid_$t *object;
status_$t *status;
```

Arguments

handle	An RPC handle.
object	The UUID of the object identified by *handle*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$inq_object` routine enables a client or server to determine the particular object that a handle represents.

If a server exports an interface through which clients can access several objects, it can use `rpc_$inq_object` to determine the object requested in a call. This routine requires an RPC handle as input, so the server can make the call only if the interface uses explicit handles (that is, if each operation in the interface has a handle parameter). If the interface uses an implicit handle, the handle identifier is not passed to the server.

Examples

A database server that manages multiple databases must determine the particular database to be accessed whenever it receives a remote procedure call. Each manager routine makes the following call; the routine then uses the returned UUID to identify the database to be accessed:

```
rpc_$inq_object (handle, &db_uuid, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_listen (3ncs)

Name

rpc_listen – listen for and handle remote procedure call (RPC) packets (server only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$listen(max_calls, status)
unsigned long max_calls;
status_$t *status;
```

Arguments

max_calls	This value indicates the maximum number of calls that the server is allowed to process concurrently. On ULTRIX systems, this value should be 1; any other value is ignored and defaulted to one.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$listen` routine dispatches incoming remote procedure call requests to manager procedures and returns the responses to the client. You must issue `rpc_$use_family` or `rpc_$use_family_wk` before you use `rpc_$listen`. This routine normally does not return. A return from this routine indicates either an irrecoverable error, or that an `rpc_shutdown` call has been issued. If `status.all` is equal to **status_$ok** , the assumption is that `rpc_$shutdown` has occurred.

Examples

Listen for incoming remote procedure call requests.

```
rpc_$listen (1, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$bad_pkt	The server or client has received an ill-formed packet.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
/usr/include/idl/c/rpc.h
```

See Also

intro(3ncs), rpc_shutdown(3ncs)

rpc_name_to_sockaddr (3ncs)

Name

rpc_name_to_sockaddr – convert a host name and port number to a socket address (client or server)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$name_to_sockaddr(name, nlength, port, family, sockaddr,
                           slength, status)
unsigned char name;
unsigned long nlength;
unsigned long port;
unsigned long family;
socket_$addr_t *sockaddr;
unsigned long *slength;
status_$t *status;
```

Arguments

name	A string that contains a host name and, optionally, a port and an address family. The format is *family*:*host*[*port*], where *family*: and [*port*] are optional. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter. The *family* part of the name parameter is **ip**; *host* is the host name; *port* is an integer port number.
nlength	The number of characters in *name*.
port	The socket port number. This parameter should have the value **rpc_$unbound_port** if you are not specifying a well-known port; in this case, the returned socket address will specify the Local Location Broker (LLB) forwarding port at *host*. If you specify the port number in the *name* parameter, this parameter is ignored.
family	The address family to use for the socket address. This value corresponds to the communications protocol used to access the socket and determines how the *sockaddr* is expressed. If you specify the address family in the *name* parameter, this parameter must have the value **socket_$unspec**.
sockaddr	The socket address corresponding to *name*, *port*, and *family*.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$name_to_sockaddr` routine provides the socket address for a socket, given the host name, the port number, and the address family.

You can specify the socket address information either as one text string in the *name* parameter or by passing each of the three elements as separate parameters(*name, port,* and *family*); in the latter case, the *name* parameter should contain only the hostname.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

NOTE

This routine has been superseded by the `socket_$from_name` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_sockaddr_to_name(3ncs), socket_from_name(3ncs)

rpc_register (3ncs)

Name

rpc_register – register an interface (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$register(*ifspec, epv, status*)
rpc_$if_spec_t **ifspec*;
rpc_$epv_t *epv*;
status_$t **status*;

Arguments

ifspec	The interface being registered.
epv	The entry point vector (EPV) for the operations in the interface. The EPV is always defined in the server stub that is generated by the NIDL compiler from an interface definition.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$register` routine registers an interface with the RPC runtime library. After an interface is registered, the RPC runtime library will pass requests for that interface to the server.

You can call `rpc_$register` several times with the same interface (for example, from various subroutines of the same server), but each call must specify the same EPV. Each registration increments a reference count for the registered interface; an equal number of `rpc_$unregister` routines are then required to unregister the interface.

Examples

The following statement registers the bank interface with the bank server host's RPC runtime library:

```
rpc_$register (&bank_$if_spec, bank_$server_epv, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.

rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.
rpc_$bad_pkt	The server or client has received an ill-formed packet.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register_mgr(3ncs), rpc_register_object(3ncs), rpc_unregister(3ncs)

rpc_register_mgr (3ncs)

Name

rpc_register_mgr – register a manager (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$register_mgr(*type, ifspec, sepv, mepv, status*)
uuid_$t **type*;
rpc_$if_spec_t **ifspec*;
rpc_$generic_epv_t *sepv*;
rpc_$mgr_epv_t *mepv*;
status_$t **status*;

Arguments

type	The UUID of the type being registered.
ifspec	The interface being registered.
sepv	The generic EPV, a vector of pointers to server stub procedures.
mepv	The manager EPV, a vector of pointers to manager procedures.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$register_mgr` routine registers the set of manager procedures that implement a specified interface for a specified type.

Servers can invoke this routine several times with the same interface (*ifspec*) and generic EPV (*sepv*) but with a different object type (*type*) and manager EPV (*mepv*) on each invocation. This technique allows a server to export several implementations of the same interface.

Servers that export several versions of the same interface (but not different implementations for different types) must also use `rpc_$register_mgr`, not `rpc_$register`. Such servers should supply **uuid_$nil** as the *type* to `rpc_$register_mgr`.

If a server uses `rpc_$register_mgr` to register a manager for a specific interface and a specific type that is not nil, the server must use `rpc_$register_object` to register an object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.

rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_object(3ncs), rpc_unregister(3ncs)

rpc_register_object (3ncs)

Name

rpc_register_object – register an object (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$register_object(*object, type, status*)
uuid_$t **object*;
uuid_$t **type*;
status_$t **status*;

Arguments

object	The UUID of the object being registered.
type	The UUID of the type of the object.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$register_object` routine declares that a server supports operations on a particular object and declares the type of that object.

A server must register objects with `rpc_$register_object` only if it registers generic interfaces with `rpc_$register_mgr`. When a server receives a call, the RPC runtime library searches for the object identified in the call (that is the object that the client specified in the handle) among the objects registered by the server. If the object is found, the type of the object determines which of the manager EPVs should be used to operate on the object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

rpc_$illegal_register You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_mgr(3ncs), rpc_unregister(3ncs)

rpc_set_async_ack (3ncs)

Name

rpc_set_async_ack – set or clear asynchronous-acknowledgement mode (client only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$set_async_ack (state)
unsigned long state;
```

Arguments

state	If "true" (nonzero), asynchronous-acknowledgement mode is set. If "false" (zero), synchronous-acknowledgement mode is set.

Description

The `rpc_$set_async_ack` call sets or clears asynchronous-acknowledgement mode in a client.

Synchronous-acknowledgement mode is the default. Calling `rpc_$set_async_ack` with a nonzero value for *state* sets asynchronous-acknowledgement mode. Calling it with a zero value for *state* sets synchronous-acknowledgement mode.

After a client makes a remote procedure call and receives a reply from a server, the RPC runtime library at the client acknowledges its receipt of the reply. This "reply acknowledgement" can occur either synchronously (before the runtime library returns to the caller) or asynchronously (after the runtime library returns to the caller).

It is generally good to allow asynchronous reply acknowledgements. Asynchronous-acknowledgement mode can save the client runtime library from making explicit reply acknowledgements, because after a client receives a reply, it may shortly issue another call that can act as an implicit acknowledgement.

Asynchronous-acknowledgement mode requires that an "alarm" be set to go off sometime after the remote procedure call returns. Unfortunately, setting the alarm can cause two problems:

1. There may be only one alarm that can be set, and the application itself may be trying to use it.
2. If, at the time the alarm goes off, the application is blocked in a system call that is doing I/O to a "slow device" (such as a terminal), the system call will return an error (with the EINTR errno); the application may not be coded to expect this error. If neither of these problems exists, the application should set asynchronous-acknowledgement mode to get greater efficiency.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_set_binding (3ncs)

Name

rpc_set_binding – bind an RPC handle to a server (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$set_binding(*handle, sockaddr, slength, status*)
handle_t *handle*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
status_$t **status*;

Arguments

handle	An RPC handle.
sockaddr	The socket address of the server with which the handle is being associated.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$set_binding` routine sets the binding of an RPC handle to the specified server. The handle then identifies a specific object at a specific server. Any subsequent remote procedure calls that a client makes using the handle are sent to this destination.

You can use this routine either to set the binding in an unbound handle or to replace the existing binding in a fully bound or bound-to-host handle.

Examples

The following statement sets the binding on the handle `h` to the first server in the `lbresults` array, which was returned by a previous Location Broker lookup routine, `lb_lookup_interface`:

```
rpc_$set_binding (h, &lbresults[0].saddr, lbresults[0].saddr_len,
                      &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock	The RPC runtime library created a socket but was unable to bind it to a socket address.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_clear_binding(3ncs), rpc_clear_server_binding(3ncs)

rpc_set_fault_mode (3ncs)

Name

rpc_set_fault_mode – set the fault-handling mode for a server (server only)

Syntax

```
#include <idl/c/rpc.h>

unsigned long rpc_$set_fault_mode(state)
unsigned long state;
```

Arguments

state If 'true' (not zero), the server exits when a fault occurs. If 'false' (zero), the server reflects faults back to the client.

Description

The `rpc_$set_fault_mode` function controls the handling of faults that occur in user server routines.

In the default mode, the server reflects faults back to the client and continues processing. Calling `rpc_$set_fault_mode` with value other than zero for *state* sets the fault-handling mode so that the server sends an **rpc_$comm_failure** fault back to the client and exits. Calling `rpc_$set_fault_mode` with *state* equal to zero resets the fault-handling mode to the default.

This function returns the previous state of the fault-handling mode.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

Name

rpc_set_short_timeout – set or clear short-timeout mode (client only)

Syntax

```
#include <idl/c/rpc.h>

unsigned long rpc_$set_short_timeout(handle, state, status)
handle_t handle;
unsigned long state;
status_$t *status;
```

Arguments

handle	An RPC handle.
on	If 'true' (not zero), short-timeout mode is set on *handle*. If 'false' (zero), standard timeouts are set.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$set_short_timeout` routine sets or clears short-timeout mode on a handle. If a client uses a handle in short-timeout mode to make a remote procedure call, but the server does not respond, the call fails quickly. As soon as the server responds, standard timeouts take effect and apply for the remainder of the call.

Calling `rpc_$set_short_timeout` with a value other than zero for *state* sets short-timeout mode. Calling it with *state* equal to zero, sets standard timeouts. Standard timeouts are the default.

This routine returns the previous setting of the timeout mode in `status.all`.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_shutdown (3ncs)

Name

rpc_shutdown – shut down a server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$shutdown(*status*)
status_$t **status*;

Arguments

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$shutdown` routine shuts down a server. When this routine is executed, the server stops processing incoming calls and `rpc_$listen` returns.

If `rpc_$shutdown` is called from within a remote procedure, that procedure completes, and the server shuts down after replying to the caller.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure The call could not be completed due to a communication problem.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_allow_remote_shutdown(3ncs), rpc_listen(3ncs), rrpc_shutdown(3ncs)

Name

rpc_sockaddr_to_name – convert a socket address to a host name and port number (client or server)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$sockaddr_to_name(sockaddr, slength, name, nlength,
                                                port, status)
socket_$addr_t *sockaddr;
unsigned long slength;
unsigned char name;
unsigned long *nlength;
unsigned long *port;
status_$t *status;
```

Arguments

sockaddr	A socket address.
slength	The length, in bytes, of *sockaddr*.
name	A string that contains the host name and the address family. The format is *family:host [port]* where *family* is **ip**.
nlength	On input, *nlength* is the length of the *name* buffer. On output, *nlength* is the number of characters returned in the *name* parameter.
port	The socket port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$sockaddr_to_name` routine provides the address family, the host name, and the port number identified by the specified socket address.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

NOTE

This routine has been superseded by the `socket_$to_name` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_name_to_sockaddr(3ncs), socket_to_name(3ncs)

Name

rpc_unregister – unregister an interface (server only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$unregister(ifspec, status)
rpc_$if_spec_t *ifspec;
status_$t *status;
```

Arguments

ifspec	An **rpc_$if_spec_t**. An interface specifier obtained from a previous RPC register call. The interface being unregistered.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$unregister` routine unregisters an interface that the server previously registered with the RPC runtime library. After an interface is unregistered, the RPC runtime library will not pass requests for that interface to the server.

If a server uses several `rpc_$register` or `rpc_$register_mgr` routines to register an interface more than once, then it must call `rpc_$unregister` an equal number of times to unregister the interface.

Examples

The following statement unregisters a matrix arithmetic interface:

```
rpc_$unregister (&matrix_$if_spec, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_mgr(3ncs), rpc_register_object(3ncs)

Name

rpc_use_family – create a socket of a specified address family for a remote procedure call (RPC) server (server only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$use_family(family, sockaddr, slength, status)
unsigned long family;
socket_$addr_t *sockaddr;
unsigned long *slength;
status_$t *status;
```

Arguments

family	The address family of the socket to be created. The value must be one of **socket_$internet** or **socket_$unspec**.
sockaddr	The socket address of the socket on which the server will listen.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$use_family` routine creates a socket for a server without specifying its port number. The RPC runtime software assigns a port number. If a server must listen on a particular well-known port, use `rpc_$use_family_wk` to create the socket.

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers should make this call once per supported address family.

Examples

The following statement creates a server's socket:

```
rpc_$use_family (family, &saddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

rpc_$too_many_sockets
: The server is trying to use more than the maximum number of sockets that is allowed; it has called `rpc_$use_family` or `rpc_$use_family_wk` too many times.

rpc_$addr_in_use
: The address and port specified in an `rpc_$use_family_wk` routine are already in use. This is caused by multiple calls to `rpc_$use_family_wk` with the same well-known port.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_use_family_wk(3ncs)

Name

rpc_use_family_wk – create a socket with a well-known port for a remote procedure call (RPC) server (server only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$use_family_wk(family, ifspec, sockaddr, slength, status)
unsigned long family;
rpc_$if_spec_t *ifspec;
socket_$addr_t *sockaddr;
unsigned long *slength;
status_$t *status;
```

Arguments

family	The address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the sockaddr is expressed. The value must be one of **socket_$unspec** or **socket_$internet.**
ifspec	The interface that will be registered by the server. Typically, this parameter is the interface *if_spec* generated by the NIDL compiler from the interface definition; the well-known port is specified as an interface attribute.
sockaddr	The socket address of the socket on which the server will listen.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rpc_$use_family_wk` routine creates a socket that uses the port specified through the *if_spec* parameter. Use this routine to create a socket only if a server must listen on a particular well-known port. Otherwise, use `rpc_$use_family`.

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers that use well-known ports should make this call once per supported address family.

Examples

The following statement creates the well-known socket identified by `sockaddr` for an array processor server:

```
rpc_$use_family_wk (socket_$internet, &matrix$if_spec,
                    &sockaddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_create_sock
: The RPC runtime library was unable to create a socket.

rpc_$not_in_call
: An internal error.

rpc_$proto_error
: An internal protocol error.

rpc_$too_many_sockets
: The server is trying to use more than the maximum number of sockets that is allowed; it has called `rpc_$use_family` or `rpc_$use_family_wk` too many times.

rpc_$bad_pkt
: The server or client has received an ill-formed packet.

rpc_$addr_in_use
: The address and port specified in an `rpc_$use_family_wk` routine are already in use. This is caused by multiple calls to `rpc_$use_family_wk` with the same well-known port.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_use_family(3ncs)

Name

rrpc_inq_interfaces – obtain a list of the interfaces that a server exports

Syntax

```
#include <idl/c/rrpc.h>

void rrpc_$inq_interfaces(handle, max_ifs, ifs, l_if, status)
handle_t handle;
unsigned long max_ifs;
rrpc_$interface_vec_t ifs[];
unsigned long *l_if;
status_$t *status;
```

Arguments

handle	An RPC handle.
max_ifs	The maximum number of elements in the array of interface specifiers.
ifs	An array of **rpc_$if_spec_t**.
l_if	The index of the last element in the returned array.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rrpc_$inq_interfaces` routine returns an array of RPC interface specifiers.

Files

```
/usr/include/idl/c/rrpc.h
/usr/include/idl/rrpc.idl
```

See Also

intro(3ncs)

rrpc_inq_stats (3ncs)

Name

rrpc_inq_stats – obtain statistics about a server

Syntax

```
#include <idl/c/rrpc.h>

void rrpc_$inq_stats(handle, max_stats, stats, l_stat, status)
handle_t handle;
unsigned long max_stats;
rrpc_$stat_vec_t stats;
unsigned long *l_stat;
status_$t *status;
```

Arguments

handle A remote procedure call (RPC) *handle* .

max_stats The maximum number of elements in the array of statistics.

stats An array of 32-bit integers representing statistics about the server. A set of **rrpc_$sv** constants defines indices for the elements in this array. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in
The number of calls processed by the server.

rrpc_$sv_rcvd
The number of packets received by the server.

rrpc_$sv_sent
The number of packets sent by the server.

rrpc_$sv_calls_out
The number of calls made by the server.

rrpc_$sv_frag_resends
The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
The number of duplicate fragments received by the server.

l_stat The index of the last element in the returned array.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `rrpc_$inq_stats` routine returns an array of integer statistics about a server.

Files

```
/usr/indlude/idl/c/rrpc.h
/usr/include/idl/rrpc.idl
```

See Also

intro(3ncs)

rrpc_shutdown(3ncs)

Name

rrpc_shutdown – shut down a server

Syntax

```
#include <idl/c/rrpc.h>

void rrpc_$shutdown(handle, status)
handle_t handle;
status_$t *status;
```

Arguments

handle A remote procedure call (RPC) handle.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rrpc_$shutdown` routine shuts down a server, if the server allows it. A server can use the `rpc_$allow_remote_shutdown` routine to allow or disallow remote shutdown.

Diagnostics

This section lists status codes for errors returned by this `rrpc_$` routine in `status.all`.

rrpc_$shutdown_not_allowd
You send an `rrpc_shutdown` request to a server that has not issued an `rpc_allow_remote_shutdown` call.

Files

```
/usr/include/idl/c/rrpc.h
/usr/include/idl/rrpc.idl
```

See Also

intro(3ncs), rpc_allow_remote_shutdown(3ncs), rpc_shutdown(3ncs)

Name

socket_equal – compare two socket addresses

Syntax

```
#include <idl/c/socket.h> .

boolean socket_$equal(sockaddr1, s1length, sockaddr2, s2length, flags,
                      status)
socket_$addr_t *sockaddr1;
unsigned long s1length;
socket_$addr_t *sockaddr2;
unsigned long s2length;
unsigned long flags;
status_$t *status;
```

Arguments

sockaddr1	A socket address. The socket address is the structure returned by either `rpc_use_family` or `rpc_use_family_wk`.
s1length	The length, in bytes, of *sockaddr1*.
sockaddr2	A socket address. The socket address is the structure returned by either `rpc_use_family` or `rpc_use_family_wk`.
s2length	The length, in bytes, of *sockaddr2*.
flags	The logical OR of values selected from the following:
	socket_$eq_hostid Indicates that the host IDs are to be compared.
	socket_$eq_netaddr Indicates that the network addresses are to be compared.
	socket_$eq_port Indicates that the port numbers are to be compared.
	socket_$eq_network Indicates that the network IDs are to be compared.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$equal` routine compares two socket addresses. The *flags* parameter determines which fields of the socket addresses are compared. The call returns 'true' (not zero) if all of the fields compared are equal, 'false' (zero) if not.

Examples

The following routine compares the network and host IDs in the socket addresses *sockaddr1* and *sockaddr2:*

```
if (socket_$equal (&sockaddr1, s1length, &sockaddr2, s2length,
        socket_$eq_network | socket_$eq_hostid, &status))
printf ("sockaddrs have equal network and host IDs\n");
```

Files

```
/usr/include/idl/c/socket.h
/usr/include/idl/socket.idl
```

See Also

intro(3ncs)

Name

socket_family_from_name – convert an address family name to an integer

Syntax

#include <idl/c/socket.h>

unsigned long socket_$family_from_name(*name, nlength, status*)
socket_$string_t *name*;
unsigned long *nlength*;
status_$t **status*;

Arguments

name	The textual name of an address family. Currently, only **ip** is supported.
nlength	The length, in bytes, of *name*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$family_from_name` routine returns the integer representation of the address family specified in the text string *name*.

Examples

The server program for the banks example, `/usr/examples/banks/bankd.c` accepts a textual family name as its first argument. The program uses the following `socket_$family_from_name` routine to convert this name to the corresponding integer representation:

```
family = socket_$family_from_name
             (argv[1], (long)strlen(argv[1]), &status);
```

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_name(3ncs)

socket_family_to_name (3ncs)

Name

socket_family_to_name – convert an integer address family to a textual name

Syntax

```
#include <idl/c/socket.h>

void socket_$family_to_name(family, name, nlength, status)
unsigned long family;
socket_$string_t name;
unsigned long *nlength;
status_$t *status;
```

Arguments

family	The integer representation of an address family.
name	The textual name of *family*. Currently, only **ip** is supported.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the returned name.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$family_to_name` routine converts the integer representation of an address family to a textual name for the family.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs)

Name

socket_from_name – convert a name and port number to a socket address

Syntax

```
#include <idl/c/socket.h>

void socket_$from_name(family, name, nlength, port, sockaddr, slength,
                       status)
unsigned long family;
socket_$string_t name;
unsigned long nlength;
unsigned long port;
socket_$addr_t *sockaddr;
unsigned long *slength;
status_$t *status;
```

Arguments

family The integer representation of an address family. Value can be **socket_$internet** or **socket_$unspec** If the *family* parameter is **socket_$unspec**, then the *name* parameter is scanned for a prefix of *family*: (for example, **ip:**).

name A string in the format *family*:*host*[*port*], where *family*:, *host*, and [*port*] are all optional.

The *family* is an address family. The only valid *family* is **ip**. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter.

The *host* is a host name. A leading number sign (#) can be used to indicate that the host name is in the standard numeric form (for example, #192.9.8.7). If *host* is omitted, the local host name is used.

The *port* is a port number. If you specify a *port* as part of the *name* parameter, the *port* parameter is ignored.

nlength The length, in bytes, of *name*.

port A port number. If you specify a port number in the *name* parameter, this parameter is ignored.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$from_name` routine converts a textual address family, host name, and port number to a socket address. The address family and the port number can be either specified as separate parameters or included in the *name* parameter.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_from_name(3ncs), socket_to_name(3ncs)

Name

socket_to_name – convert a socket address to a name and port number

Syntax

#include <idl/c/socket.h>

void socket_$to_name(*sockaddr, slength, name, nlength, port, status*)
socket_$addr_t **sockaddr*;
unsigned long *slength*;
socket_$string_t *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name; *host* may be in the standard numeric form (for example, #192.1.2.3) if a textual host name cannot be obtained. Currently, only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$to_name` routine converts a socket address to a textual address family, host name, and port number.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_numeric_name(3ncs)

socket_to_numeric_name (3ncs)

Name

socket_to_numeric_name – convert a socket address to a numeric name and port number

Syntax

```
#include <idl/c/socket.h>

void socket_$to_numeric_name(sockaddr, slength, name, nlength, port,
                             status

socket_$addr_t *sockaddr;
unsigned long slength;
socket_$string_t name;
unsigned long *nlength;
unsigned long *port;
status_$t *status;
```

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name in the standard numeric form (for example, #192.7.8.9 for an IP address). Currently only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned. (error if less than size of "nnnnn.nnnn"). On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok ,** then the routine that supplied it was successful.

Description

The `socket_$to_numeric_name` routine converts a socket address to a textual address family, a numeric host name, and a port number.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_name(3ncs)

socket_valid_families (3ncs)

Name

socket_valid_families – obtain a list of valid address families

Syntax

```
#include <idl/c/socket.h>

void socket_$valid_families(max_families, families, status)
unsigned long *max_families;
socket_$addr_family_t families[ ];
status_$t *status;
```

Arguments

max_families	The maximum number of families that can be returned.
families[ ]	An array of **socket_$addr_family_t**. Possible values for this type are enumerated in `/usr/include/idl/nbase.idl`. Currently, only **ip** is supported for *family*.
status	The completion status. This variable is set if the *families[ ]* array is not long enough to hold all the valid families. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$valid_families` routine returns a list of the address families that are valid on the calling host.

Examples

The following routine returns the valid address family:

```
socket_$valid_families (1, &families, $status);
```

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_valid_family(3ncs)

Name

socket_valid_family – check whether an address family is valid

Syntax

```
#include <idl/c/socket.h>

boolean socket_$valid_family(family, status)
unsigned long family;
fBstatus_$t *status;
```

Arguments

family	The integer representation of an address family.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$valid_family` routine returns 'true' if the specified address family is valid for the calling host, 'false' if not valid.

Examples

The following routine checks whether **socket_$internet** is a valid address family:

```
internetvalid = socket_$valid_family(socket_$internet, &status);
```

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_valid_families(3ncs)

uuid_decode(3ncs)

Name

uuid_decode – convert a character-string representation of a UUID into a UUID structure

Syntax

```
#include <idl/c/uuid.h>

void uuid_$decode(s, uuid, status)
uuid_$string_t s;
uuid_$t *uuid;
status_$t *status;
```

Arguments

s	The character-string representation of a UUID.
uuid	The UUID that corresponds to *s*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `uuid_$decode` routine returns the UUID corresponding to a valid character-string representation of a UUID.

Examples

The following routine returns as **foo_uuid** the UUID corresponding to the character-string representation in **foo_uuid_rep:**

```
uuid_$decode (foo_uuid_rep, &foo_uuid, &status);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs), uuid_encode(3ncs)

Name

uuid_encode – convert a UUID into its character-string representation

Syntax

```
#include <idl/c/uuid.h>

void uuid_$encode(uuid, s)
uuid_$t *uuid;
uuid_$string_t s;
```

Arguments

uuid	A UUID.
s	The character-string representation of *uuid*.

Description

The `uuid_$encode` routine returns the character-string representation of a UUID.

Examples

The following routine returns as **foo_uuid_rep** the character-string representation for the UUID **foo_uuid:**

```
uuid_$encode (&foo_uuid, foo_uuid_rep);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs), uuid_decode(3ncs)

uuid_equal(3ncs)

Name

uuid_equal – compare two UUIDs

Syntax

#include <idl/c/uuid.h>

boolean uuid_$equal(*u1*, *u2*)
uuid_$t **u1*;
uuid_$t **u2*;

Arguments

u1	A UUID.
u2	Another UUID.

Description

The `uuid_$encode` routine compares the UUIDs *u1* and *u2*. It returns 'true' if they are equal, 'false' if they are not.

Examples

The following code compares the UUIDs **bar_uuid** and **foo_uuid:**

```
if (uuid_$equal (&bar_uuid, &foo_uuid))
    printf ("bar and foo UUIDs are equal\n");
else
    printf ("bar and foo UUIDs are not equal\n");
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs)

Name

uuid_gen – generate a new UUID

Syntax

```
#include <idl/c/uuid.h>

void uuid_$gen(uuid)
uuid_$t *uuid;
```

Arguments

uuid A pointer to a UUID structure to be filled in.

Description

The `uuid_$gen` routine returns a new UUID. Typically used when creating a new remote application.

Examples

The following routine returns as **new_uuid** a new UUID:

```
uuid_$gen (&new_uuid);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs)